Extracted from:

Node.js the Right Way

Practical, Server-Side JavaScript That Scales

The Pragmatic Bookshelf

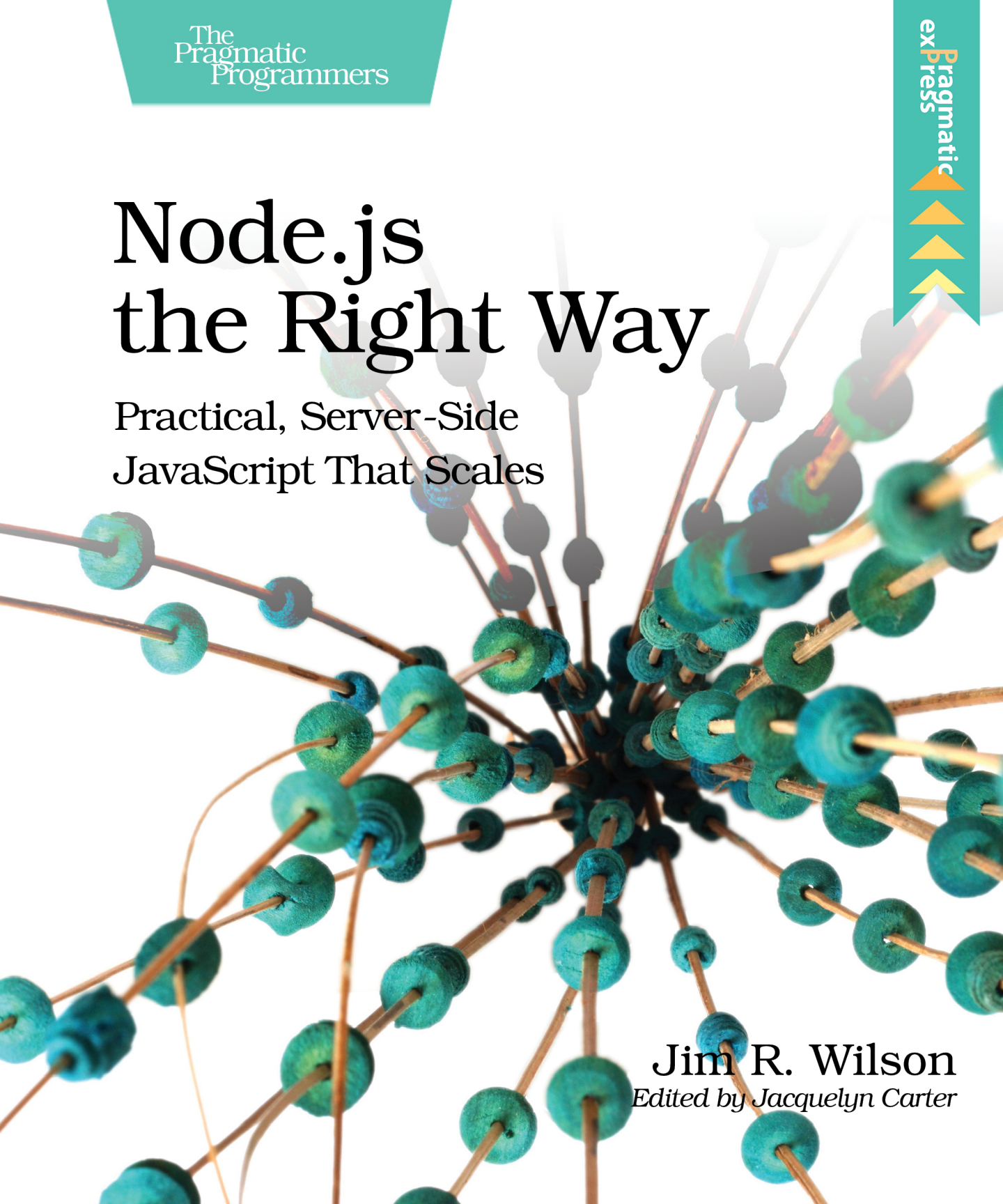
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Node.js the Right Way

Practical, Server-Side JavaScript That Scales



Jim R. Wilson

Edited by Jacquelyn Carter

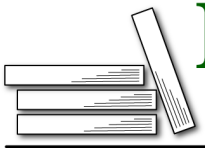
Node.js the Right Way

Practical, Server-Side JavaScript That Scales

Jim R. Wilson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-73-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2013

By now, you've mastered several approaches to handling asynchronous JavaScript code. You know how to use and write RESTful web services. You understand messaging patterns, and how to use Express.

With all of that knowledge and experience, we're now in position to develop a web application. This will take us through the following Node.js aspects:

Architecture and Core
: At this stage you've moved past the Node.js core in many respects. Node is the underlying technology that's letting you reach beyond.

Patterns
: You'll dive deeper into Express middleware, using it to implement a custom authentication handler. We'll use passport—an Express plug-in—so that users of our application can log in with their Google accounts. The client-side code uses *model-view-controller* (MVC) to separate concerns.

JavaScriptisms
: Although the Node code in this book takes advantage of the latest available ECMAScript Harmony features, those features aren't available in many web browsers. You'll learn some differences between ECMAScript 5 and ECMAScript 6 as we code JavaScript for the browser. You'll also use jQuery to perform a variety of client-side operations.

Supporting Code
: Just like npm makes it easy to pull in Node modules, *Bower* is used to manage client-side dependencies. We'll use Bower to pull in jQuery, Bootstrap, Handlebars, and others. We'll also use Redis, a fast key-value store, for managing session data.

Developing web applications is an *enormous* topic. You could fill a library with books on all the myriad concepts you need to know to master it. I've been doing it for over a decade, and I'm still learning every day.

So instead of trying to explore everything there is to know, we'll focus on what's specific to Node.js. The code download that accompanies this book contains a `web-app` directory, which has all the code from this chapter and more. I encourage you to go grab it if you haven't already, since we won't cover everything that's in there.

The application `b4` builds on the web services that we developed last chapter. This means you'll need CouchDB running, along with the data from Chapter 5, *Accessing Databases*, on page ?.

Once you have those prerequisites, let's dive in!

Storing Express Sessions in Redis

Whenever you have a user-facing web application, you'll almost always use sessions. A *session* is data that's attached to a particular user. As the user browses pages on the site or uses the web application, the server keeps track of the user through a *session cookie*. On each request, the server reads the cookie, retrieves the session data, then uses it when generating a response.

Exactly where you store this session data is up to you. By default, Express will keep the data in memory, but this doesn't readily scale. Once you have more than one Node.js process, the session data should really be stored to be in a shared place. That way, no matter which process services a user's request, it will have the correct information.

Let's see how to enable sessions in Express, and how to use Redis for storing the session data.

Enabling Sessions in Express

To enable sessions, add the `cookieParser` and `session` middleware to your `app`:

```
app.use(express.cookieParser());
app.use(express.session({ secret: 'unguessable' }));
```

The `cookieParser` middleware is responsible for parsing incoming cookies from the client, and the `session` middleware stores the session data attached to the cookie. The `secret` parameter is necessary to prevent cookie tampering and spoofing—set it to something unique to your application. Express session cookies are signed with this secret string.

Using Redis to Store Session Data

Redis is an extremely fast key/value store with tunable durability. This means you can keep it fast but risk losing data if the server were to crash, or you can sacrifice speed for greater durability guarantees.

By default, Redis keeps data in memory and then periodically writes it to disk. This makes it blazingly fast, but disaster-prone since a process crash would mean lost data.

Its speed makes Redis an ideal database for storing session data. If the server tips over, then the sessions might be lost, but at worst this means that your users will have to log in again.

To use Redis with your Node/Express app, first you'll have to install it. Installing Redis differs by platform, but once you have it installed, starting it up is a single command:

```
$ redis-server
...
[8344] 10 Sep 20:38:41.564 # Server started, Redis version 2.6.13
```

Using Redis with Express takes a couple of npm modules: redis and connect-redis. Run npm install --save to get them and add them to your package.json.

As an experiment, let's modify the server.js file from the last chapter's "Hello World" app to use Redis for session storage. Open the server.js file and add this to the const declarations at the top:

web-app/hello/server.js

```
redisClient = require('redis').createClient(),
RedisStore = require('connect-redis')(express),
```

The first line constructs a client for the Redis database. This will immediately open a TCP socket to your Redis server. The second line produces a class you can use to instantiate a Redis-based backing store for sessions.

Finally, in the middleware section of your server.js file, use the cookie and session middleware with Redis like so:

web-app/hello/server.js

```
app.use(express.cookieParser());
app.use(express.session({
  secret: 'unguessable',
  store: new RedisStore({
    client: redisClient
  })
}));

app.get('/api/:name', function(req, res) {
  res.json(200, { "hello": req.params.name });
});

app.listen(3000, function(){
  console.log("ready captain.");
});
```

Here we create a new instance of RedisStore based on our redisClient for the session middleware to use. With those changes in place, start up the server:

```
$ npm start

> hello@0.1.0 start ./hello
> node --harmony ./server.js

ready captain.
```

Then, in a second terminal, let's check out the HTTP headers with curl:

```
$ curl -i -X HEAD http://localhost:3000/api/test
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 21
Set-Cookie: connect.sid=s%3A-Coz4SVOKykt-fb5bTv7CHOj.IKL445JyFaKDq2aJ2%2FC95dh
            87pgElsNmc2mq3mpbjNE; Path=/; HttpOnly
Date: Fri, 27 Sep 2013 22:03:52 GMT
Connection: keep-alive
```

Notice the long Set-Cookie header—this is the session cookie. You can confirm that the data has been saved in Redis with the redis-cli command-line tool:

```
$ redis-cli KEYS 'sess:*'
1) "sess:-Coz4SVOKykt-fb5bTv7CHOj"
```

Here we've asked for all the keys in Redis that start with sess:—that is, the stored session data. Typically you won't go digging around in Redis for this data yourself, but it's helpful to confirm that everything is working.

Let's return now to the b4 application, which will use Redis for session storage in just this way. With Redis running, we can build a stateful web application on top our RESTful services.

Creating a Single-Page Web Application

A single-page web application consists of an HTML page, CSS to style it, and JavaScript to perform the application logic. All of these things can be delivered as static files, meaning they don't require any special server-side processing. You can just serve them up from the file system.

In this section, you'll see how to serve static content from Express alongside your RESTful APIs. Some of the static content will come from third-party dependencies, which we'll pull from Bower.

Serving Static Content with Express

Express comes with a convenient way to serve static files. All you have to do is use the express.static middleware and provide a directory. For example, these lines appear in the b4 server:

web-app/b4/server.js

```
app.use(express.static(__dirname + '/static'));
app.use(express.static(__dirname + '/bower_components'));
```

These two lines tell Express to serve static content out of the static/ and bower_components/ directories of the project. This means that if Express can't find

a particular route, it'll fall back to serving the static content, checking these directories one at a time.

The static middleware is special in this regard. Most of the time, middleware has its effect in the middle of the request processing, but static appends its effect to the end of the chain (even after the route handlers have run).

For instance, the project's static/ directory contains three files: index.html, app.css, and app.js. These contain the HTML, CSS, and client-side JavaScript for the b4 application, respectively.

With the server running, when you request http://localhost:3000/index.html, Express will serve up static/index.html because we don't have an explicit route for it.

Installing Bower Components

Bower is a package manager for front-end code, like JavaScript libraries. You install Bower components in much the same way you install npm modules.

Here's the bower.json file from the b4 application:

web-app/b4/bower.json

```
{
  "name": "b4",
  "version": "0.0.1",
  "dependencies": {
    "jquery": "~2.0.3",
    "bootstrap": "~3.0.0",
    "typeahead.js": "~0.9.3",
    "typeahead.js-bootstrap.css": "*",
    "handlebars": "~1.0.0"
  }
}
```

Like a package.json file, it contains a list of dependencies. Our app depends on these front-end packages:

- jquery—Extraordinarily popular JavaScript library for Ajax and DOM manipulation
- bootstrap—Pretty CSS styles for web applications
- typeahead.js—JavaScript library for autocompleting text fields
- typeahead.js-bootstrap.css—Bootstrap-compatible CSS for typeahead.js
- handlebars—HTML templating library

To install these packages, first install Bower through npm (if you haven't already):

```
$ npm install -g bower
```

Then install the Bower components:

```
$ bower install
```

Now you should have a bower_components/ directory containing all of the above.

Structuring a Single-Page App

With the server ready to serve the static content, and Bower components in place, we can put together the main entry point of our single-page app: the index.html file. Here's what the top of that file contains:

web-app/b4/static/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>b4 - The Better Book Bundle Builder</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="bootstrap/dist/css/bootstrap.min.css">
  <link rel="stylesheet" href="bootstrap/dist/css/bootstrap-theme.min.css">
  <link rel="stylesheet"
        href="typeahead.js-bootstrap.css/typeahead.js-bootstrap.css">
  <link rel="stylesheet" href="app.css">
</head>
<body class="container">
```

The first line—<!doctype html>—tells the browser that this is an HTML5 document. Then we enter the document's <head> section, which contains the title, some <meta> tags, and a bunch of style sheets.

The viewport <meta> tag tells mobile devices that the width of the page should match the device width (e.g., 320px on an iPhone) instead of the default, which may be much larger. It also tells the device to start off at a scale of 1.0, meaning unzoomed.

The style sheets correspond to our dependencies; first the Bootstrap core and theme, then the typeahead styles, and finally our own app.css file. The app.css file comes last, so we can easily override any styles that may have come before.

The container class on the <body> element is a Bootstrapism. It tells Bootstrap to automatically grow or shrink the width of the body so it looks good at a variety of screen sizes.

Next, let's check out the bottom of index.html:

web-app/b4/static/index.html

```
  <script src="jquery/jquery.min.js"></script>
```

```
  <script src="bootstrap/dist/js/bootstrap.min.js"></script>
  <script src="typeahead.js/dist/typeahead.min.js"></script>
  <script src="handlebars/handlebars.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

Like the head, this is where we bring in a bunch of dependencies and our own code. app.js is last so we can use all the other libraries in our application logic.

The vast middle of the index.html file contains the visual content of the application, such as templates for rendering views. We'll get to this, but before we do, we need a few more APIs.

Specifically, we need authenticated endpoints for doing user-specific actions. We'll develop those next, then return to the client side to pull it all together.